SSCM Exercise 1

Nikolaus Czernin - 11721138

Prepearation

```r
set.seed(11721138)
x1 <- rnorm(100)
x2 <- rnorm(100, mean=1000000)

x1 %>% head()
```

```
## [1]  0.5102299  0.8350386 -0.0308152 -1.7233521 -1.0009368  2.2627709
```

```r
x2 %>% head()
```

```
## [1]  999998.9  999998.6  999999.9  999999.7 1000000.1  999999.2
```

I first set the seed to be my matriculation number. The first vector contains 100 normally distributed numbers with a mean of 0 and a standard deviation of 1. The first vector contains 100 normally distributed numbers with a mean of 1000000 and a standard deviation of 1. I display both vectors' first values using `head()`.

Variance Algorithms

Algorithm 1

```r
variance_1 <- function(x, c=NULL){
  x_n <- length(x)
  x_mean <- mean(x)
  x_deviants <- x - x_mean
  x_deviants_squared <- x_deviants^2
  x_variance <- (sum(x_deviants_squared)) / (x_n - 1)
  return (x_variance)
}

variance_1(x1)
```

```
## [1] 1.0674
```

Algorithm 1 first computes the n, the length of the sample, and the mean. Then it subtracts the mean from each value, computing the deviants. The sum of the deviants is then summed up, squared, and divided by n-1. This yields the variance.

Algorithm 2

```r
variance_2 <- function(x, c=NULL){
  x_n <- length(x)
  p1 <- sum(x^2)
  p2 <- (sum(x)^2) / x_n
  x_variance <- (p1 - p2) / (x_n - 1)
  return (x_variance)
}

variance_2(x1)
```

```
## [1] 1.0674
```

Algorithm 2 also starts by computing the vector length. It then computes p1, the sum of squares. Then it computes p2, the mean of the squared sum of values. The variance is then computed by dividing the difference between p1 and p2 by n-1.

Algorithm 3

```r
variance_3 <- function(x, c=NULL){
  if (is.null(c)) c <- x[1]
  x_n <- length(x)
  p1 <- sum((x - c)^2)
  p2 <- (sum(x-c)^2)/x_n
  x_variance <- (p1 - p2) / (x_n - 1)
  return (x_variance)
}

cs <- c(0, x1[1], x1[2], x1[3], 1, 10, 100, -100, 1000, 1000000, 11721138, -11721138)
vcs <- sapply(cs, function(c) variance_3(x1, c))
data.frame(c=cs, `Variance per algorithm 3`=vcs) %>% kable()
```

c	Variance.per.algorithm.3
0.000000e+00	1.067400
5.102299e-01	1.067400
8.350386e-01	1.067400
-3.081520e-02	1.067400
1.000000e+00	1.067400
1.000000e+01	1.067400
1.000000e+02	1.067400
-1.000000e+02	1.067400
1.000000e+03	1.067400
1.000000e+06	1.067393
1.172114e+07	1.050505
-1.172114e+07	1.090909

Algorithm 3 does the same thing as algorithm 2, but every value gets shifted by a single constant `c`. Generally, since the shifting variable c is constant, the variance should stay the same no matter what value c takes …

it is scale-invariant. Computationally, this is not happening. As seen in the table above, above some high absolute number value, internal rounding errors will start changing the variance.

Algorithm 4

```r
online_variance_4 <- function(x){
  # compute the online/cumulative means
  x_online_means <- cumsum(x) / seq_along(x)

  # prepare an empty vector for the online variances
  x_online_variances <- c()

  # iterate over 1 through n
  for (i in seq(x)) {
    # the first value is alone and therefore has a variance of 0
    if (i == 1) {
      x_online_variances[i] <- 0
      next
    }
    # otherwise compute the formula
    left_term <- (i - 2) / (i - 1) * x_online_variances[i-1]
    right_term <- (x[i] - x_online_means[i-1])^2 / i
    x_online_variances[i] <- left_term + right_term
  }

  return (x_online_variances)
}

variance_4 <- function(x, c=NULL) online_variance_4(x) %>% last()

data.frame(
  n = 1:100,
  value = x1,
  Online.Variances = online_variance_4(x1)
) %>%
  filter((n < 6) | (n > 95)) %>%
  kable()
```

n	value	Online.Variances
1	0.5102299	0.0000000
2	0.8350386	0.0527503
3	-0.0308152	0.1913222
4	-1.7233521	1.2955722
5	-1.0009368	1.1332158
96	-0.9223936	1.1086453
97	-0.3359270	1.0974765
98	-0.6180275	1.0884358
99	0.0597081	1.0777770
100	0.0771235	1.0674004

Algorithm 4 is an online computation, which computes the variances anew for every element in a sequence when an element is added. Firstly, this requires the online means, which are the cumulative means of the vector. This is computed using `cumsum()` and `seq_along()`. I then iterate over the values `1, 2, 3, ..., n-2, n-1, n`. In the first iteration, the variance is zero, because 1 single element has a variance of 0. The following iterations then access the means and variances of the previous iterations. In every following iteration I compute the left term, which accesses the saved variances of the respective previous iteration, and the right term of the formula, which accesses the new/current element in the sequence and deducts the mean of the previous iteration.

Looking at the table above, you can observe the variance being zero at first and converging to 1 over time, and finally ending at about the same value as computed in the previous 3 algorithms.

Wrapper function

```r
get_variances <- function(x, c=NULL){

  funs <- c(var,   variance_1,   variance_2,   variance_3,   variance_4)
  names <- c("Base R", "Algorithm 1", "Algorithm 2", "Algorithm 3", "Algorithm 4 (online)")

  out <- data.frame(
    Function = names,
    Variance = sapply(funs, function(f) f(x, c)),
    Mean.Time.ms = sapply(funs, function(f) mean(microbenchmark(f(x))$time) / 1e6)
  )
  return (out)
}
```

I put the `microbenchmark` function inside the wrapper function. It calls the base-R `var()` function, as well as all 4 of my own variance functions. From the online variance algorithm #4, it saves only the last value. The names of the algorithms and the resulting variances are output in a single dataframe. Computation times are added to the resulting dataframe.

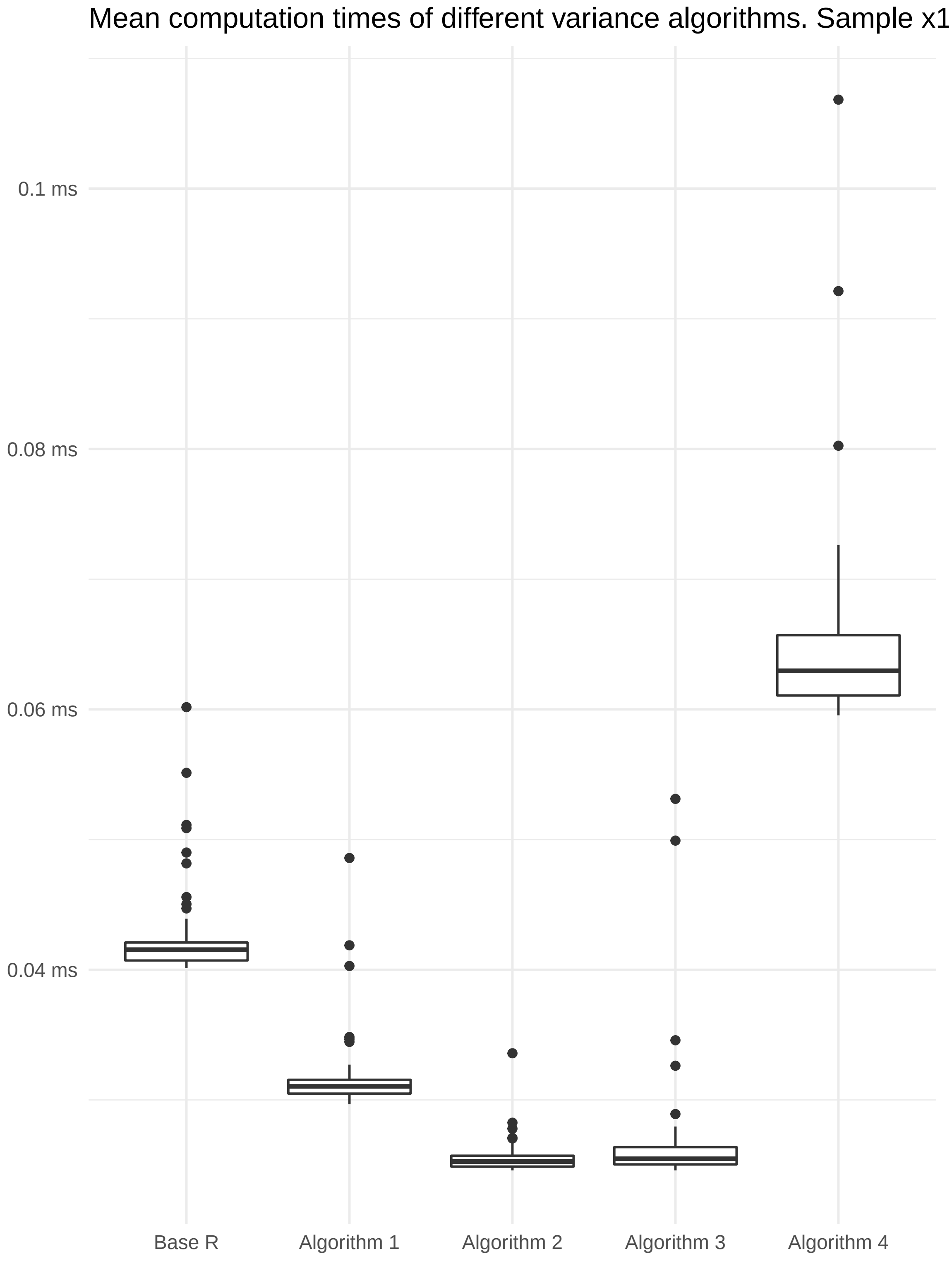
Comparing Base R and custom function computation time

Sample 1

```r
get_variances(x1) %>% kable()
```

Function	Variance	Mean.Time.ms
Base R	1.0674	0.0438423
Algorithm 1	1.0674	0.0368615
Algorithm 2	1.0674	0.0296481
Algorithm 3	1.0674	0.0281474
Algorithm 4 (online)	1.0674	0.0768085

```r
microbenchmark(
  "Base R" = var(x1),
  "Algorithm 1" = variance_1(x1),
  "Algorithm 2" = variance_2(x1),
  "Algorithm 3" = variance_3(x1),
  "Algorithm 4" = variance_4(x1)
) %>%
  ggplot(aes(x = expr, y = time / 1e6)) +
  geom_boxplot() +
  theme_minimal() +
  theme(
    axis.title = element_blank(),
  ) +
  scale_y_continuous(labels = function(x) paste0(x, " ms")) +
  ggtitle("Mean computation times of different variance algorithms. Sample x1")
```

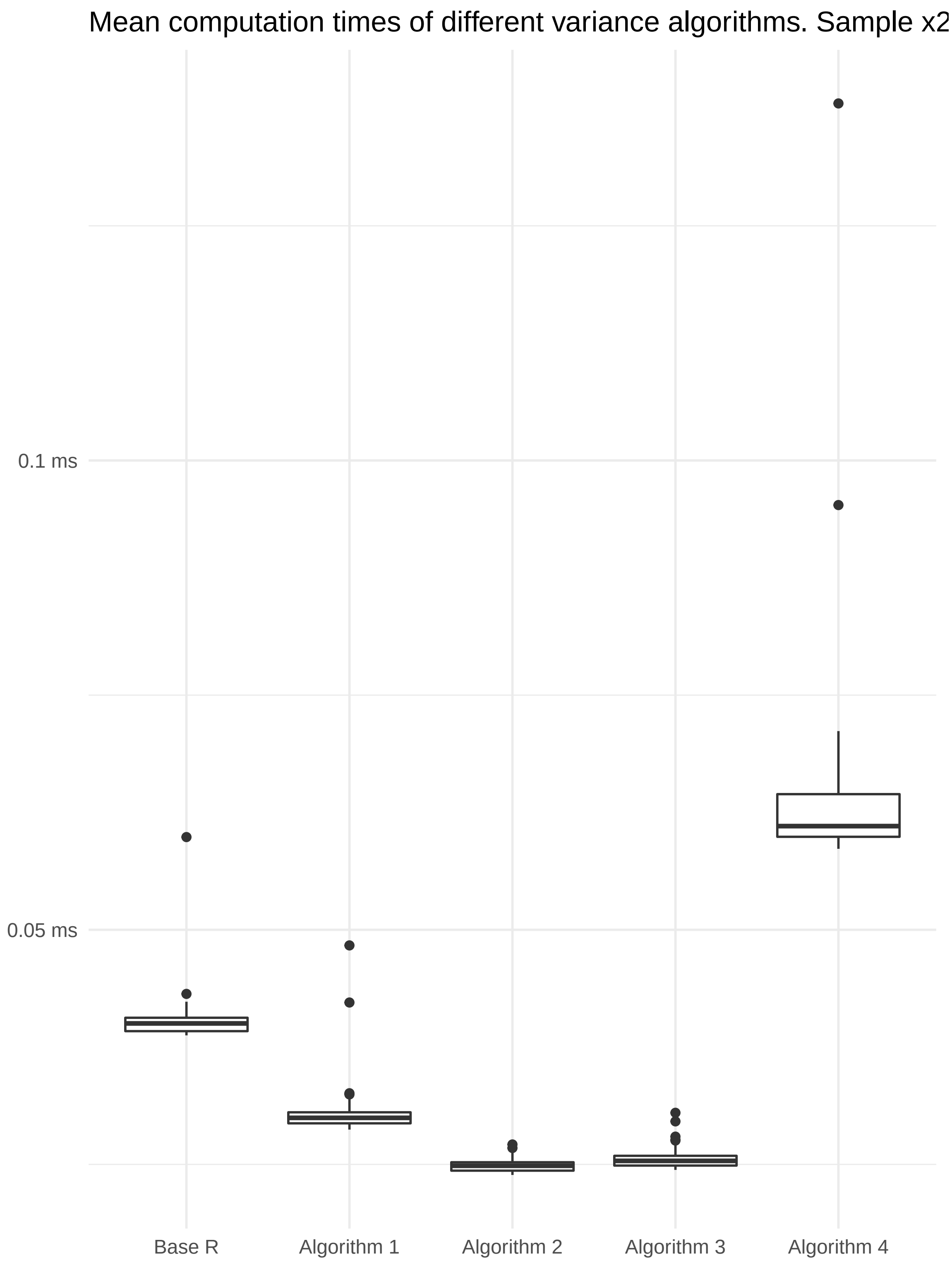

Mean computation times of different variance algorithms. Sample x1
0.1 ms
0.08 ms
0.06 ms
0.04 ms
Base R
Algorithm 1
Algorithm 2
Algorithm 3
Algorithm 4

Sample 2

```r
get_variances(x1) %>% kable()
```

Function	Variance	Mean.Time.ms
Base R	1.0674	0.0548286
Algorithm 1	1.0674	0.0328427
Algorithm 2	1.0674	0.0268253
Algorithm 3	1.0674	0.0267840
Algorithm 4 (online)	1.0674	0.0670411

```r
microbenchmark(
  "Base R" = var(x2),
  "Algorithm 1" = variance_1(x2),
  "Algorithm 2" = variance_2(x2),
  "Algorithm 3" = variance_3(x2),
  "Algorithm 4" = variance_4(x2)
) %>%
  ggplot(aes(x = expr, y = time / 1e6)) +
  geom_boxplot() +
  theme_minimal() +
  theme(
    axis.title = element_blank(),
  ) +
  scale_y_continuous(labels = function(x) paste0(x, " ms")) +
  ggtitle("Mean computation times of different variance algorithms. Sample x2")
```



I get the computation times again usign `microbenchmark` and plot the `time` data in miliseconds on a `ggplot` boxplot. The table and plot show that the Base R function is in fact slower on average than the 3 custom batch variance functions. The online function is of course slower, as it computes 100 different variances sequentially.

Scale Invariance

Extra dataset

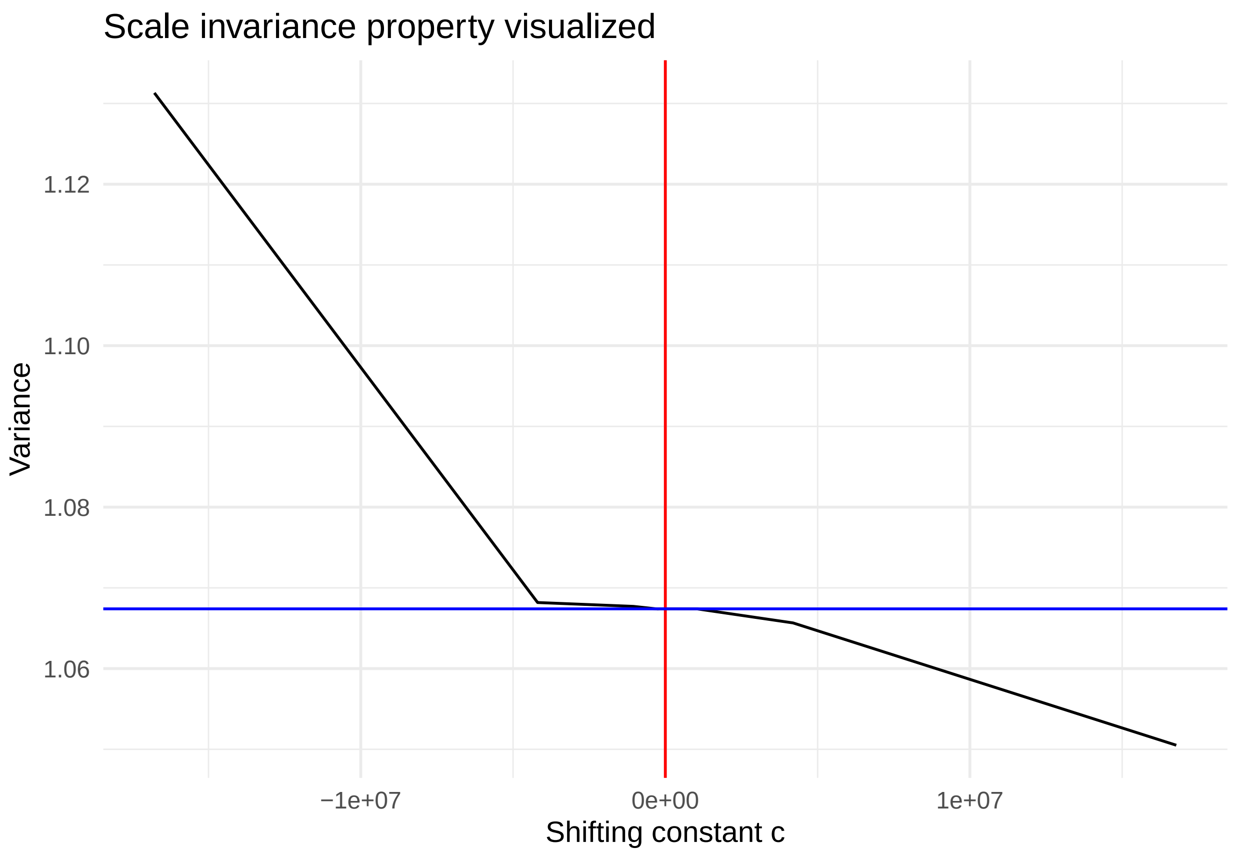
```
x3 <- rnorm(100, mean=1000000 , sd=0.0000001)
```

The third sample has the same mean as the second one, but with a very small standard deviation.

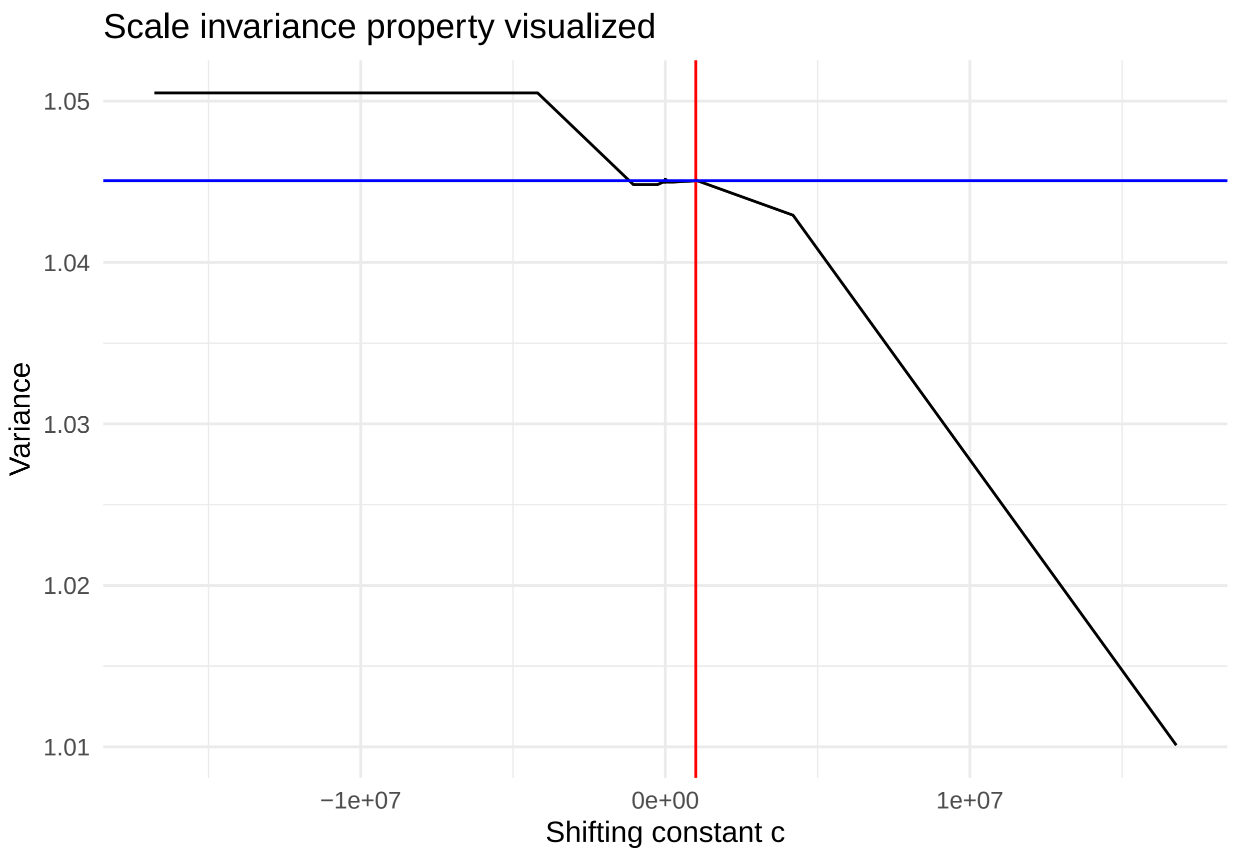
```
# x: vector of numbers
# k: number of values * 2 + 1
# j: base of exponentials
get_large_value_range <- function(x, k=12, j=2){
  xhat <- x %>% mean()
  # get exemplary values for the constant c
  constants = j ^ seq(-k, k)
  c(-1*constants, constants) + xhat
}
```

This function returns a vector of constant values centered around the mean of the given function, with exponentially rising distances between the numbers.

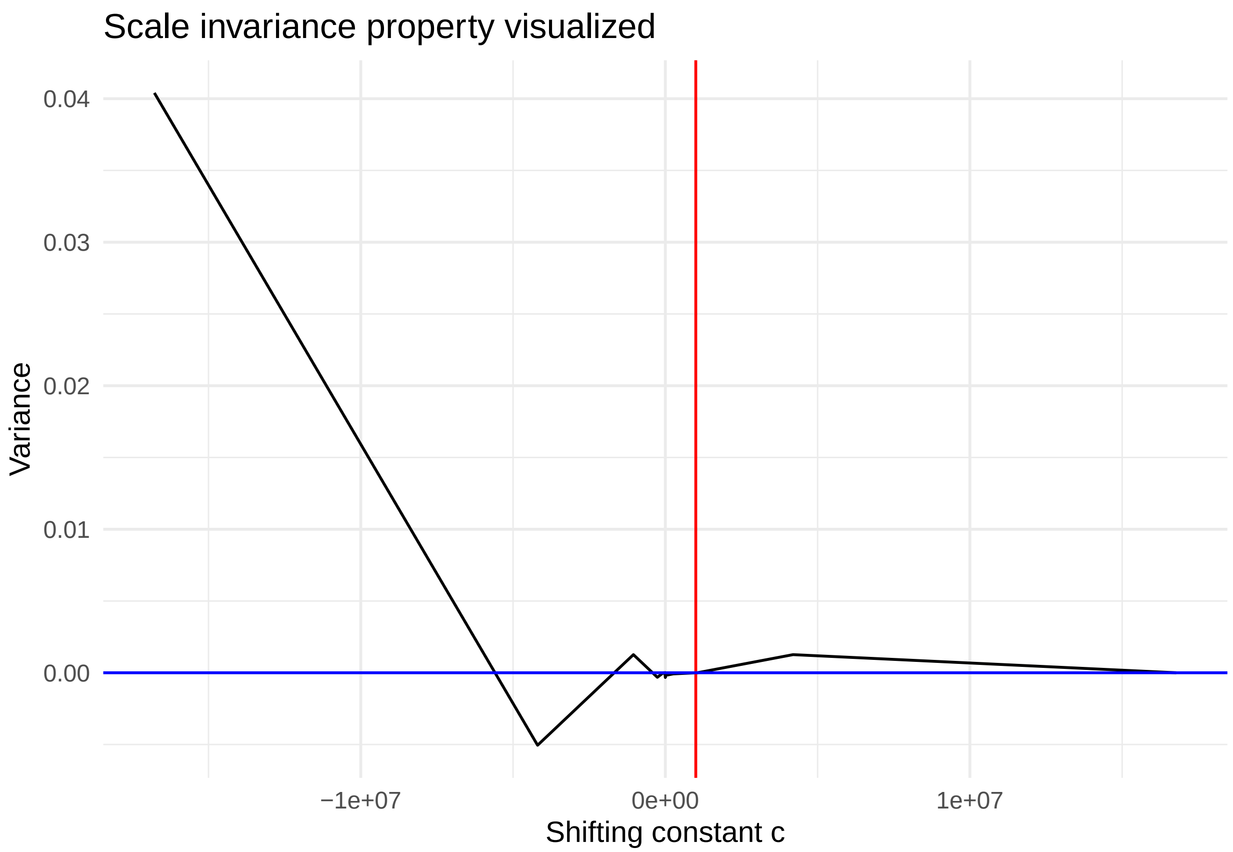
```
# Display shifted variance constant size influence on variance
foo <- function(x){
  # get exemplary values for the constant c
  constants <- get_large_value_range(x1, j=4)
  variances <- sapply(constants, function(c) variance_3(x, c))
  data.frame(
    mean=round(mean(x)),
    sd=round(sd(x), 6),
    c=constants,
    variances=variances
  ) %>%
    ggplot(aes(x=c, y=variances)) +
    geom_line() +
    geom_vline(xintercept=mean(x), color="red") +
    geom_hline(yintercept=variance_1(x), color="blue") +
    theme_minimal() +
    xlab("Shifting constant c") +
    ylab("Variance") +
    ggtitle("Scale invariance property visualized")
}

foo(x1)
```


Scale invariance property visualized
Variance
1.12
1.10
1.08
1.06
−1e+07
0e+00
1e+07
Shifting constant c

```r
foo(x2)
```


Scale invariance property visualized
1.05
1.04
1.03
1.02
1.01
Variance
−1e+07
0e+00
1e+07
Shifting constant c

```
foo(x3)
```



The plots above display the influence of the shifting constant on the resulting variances. The scale invariance property appears not to hold for larger absolute values of c. The red lines mark a c value equal to the sample mean, the blue lines indicate the variance of a shift of zero.

Condition Number

```
con_num_variance <- function(x) sqrt(sum(x^2) / sum((x - mean(x))^2))

con_num_variance(rnorm(100, mean = 0, sd=1))
```

```
## [1] 1.003351
```

```
con_num_variance(rnorm(100, mean = 1e-6, sd=1))
```

```
## [1] 1.00122
```

```
con_num_variance(rnorm(100, mean = 1e-6, sd=1e-3))
```

```
## [1] 1.000052
```

On the first sample with a mean of 0 or very close to 0 and a standard deviation of 1 or smaller the condition number is converging to 0.

```r
con_num_variance(rnorm(100, mean = 1e6, sd=1e6))
```

```
## [1] 1.419964
```

Even when using a very high mean and an equally high standard deviation, the condition number remains close to 1.

```r
x4 <- rnorm(100, mean = 1e6, sd=1e-6)

var(x4)
```

```
## [1] 9.196747e-13
```

```r
con_num_variance(x4)
```

```
## [1] 1.04801e+12
```

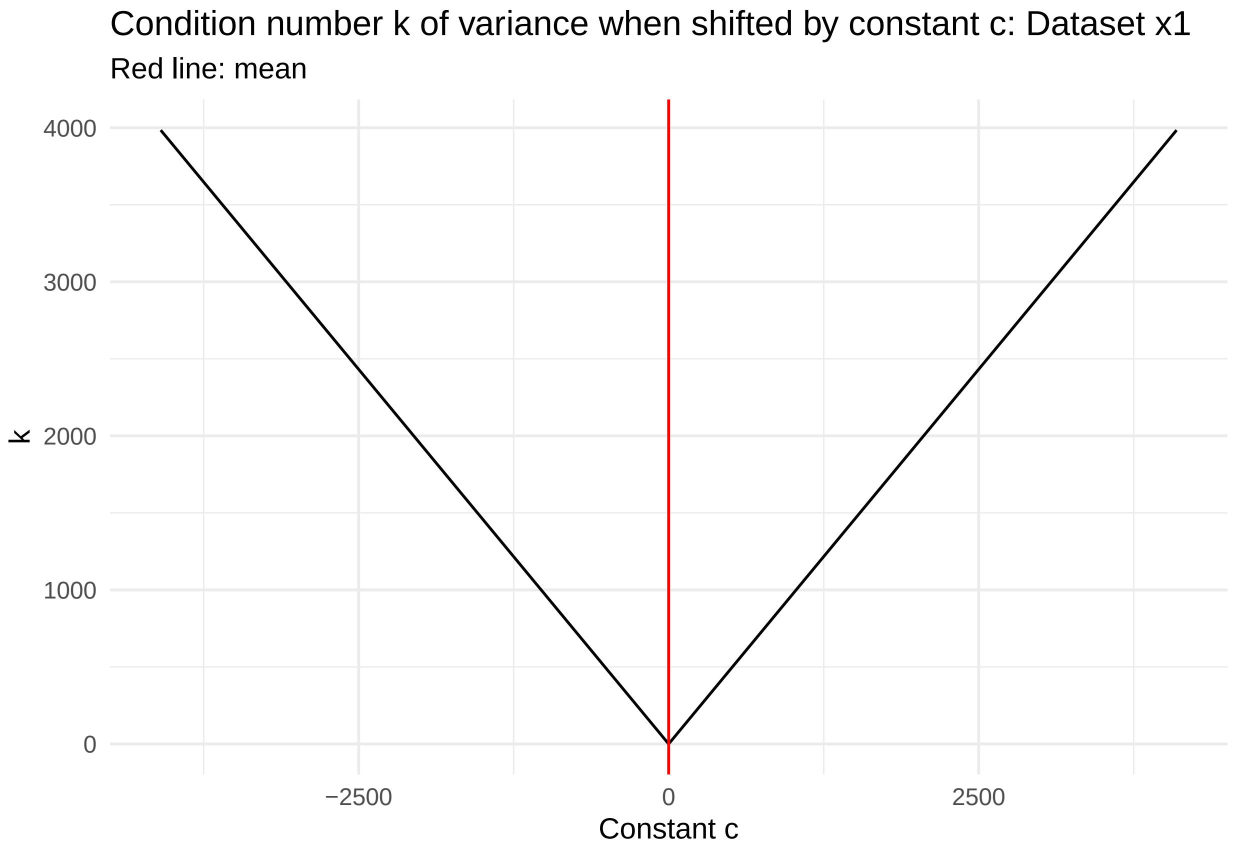
When using a high mean and a low standard deviation on the other hand, the condition number is very large, even though the variance is very low.

Shifted variance condition number

```r
con_num_variance_shift <- function(x, c) sqrt(1 + length(x) / sum((x-mean(x))^2) * (mean(x)-c)^2)

# visualize condition number of variance with shifting constant
show_shifted_k <- function(x, title_append=""){
  # the the sample mean
  xhat <- x %>% mean()
  # get exemplary values for the constant c
  constants = 2 ^ seq(-12, 12)
  constants = c(-1*constants, constants) + xhat
  # create a dataframe to viszalize everything
  data.frame(
    cs = constants,
    vs = sapply(constants, function(c) variance_3(x, c)),
    ks = sapply(constants, function(c) con_num_variance_shift(x, c))
  ) %>%
    ggplot(aes(x=cs, y=ks)) +
    geom_line() +
    geom_vline(xintercept = xhat, color="red") +
    theme_minimal() +
    labs(x="Constant c", y="k", subtitle="Red line: mean") +
    ggtitle("Condition number k of variance when shifted by constant c" %>% paste0(title_append))
}

con_num_variance_shift(x1, 0)
```

```
## [1] 1.010097
```

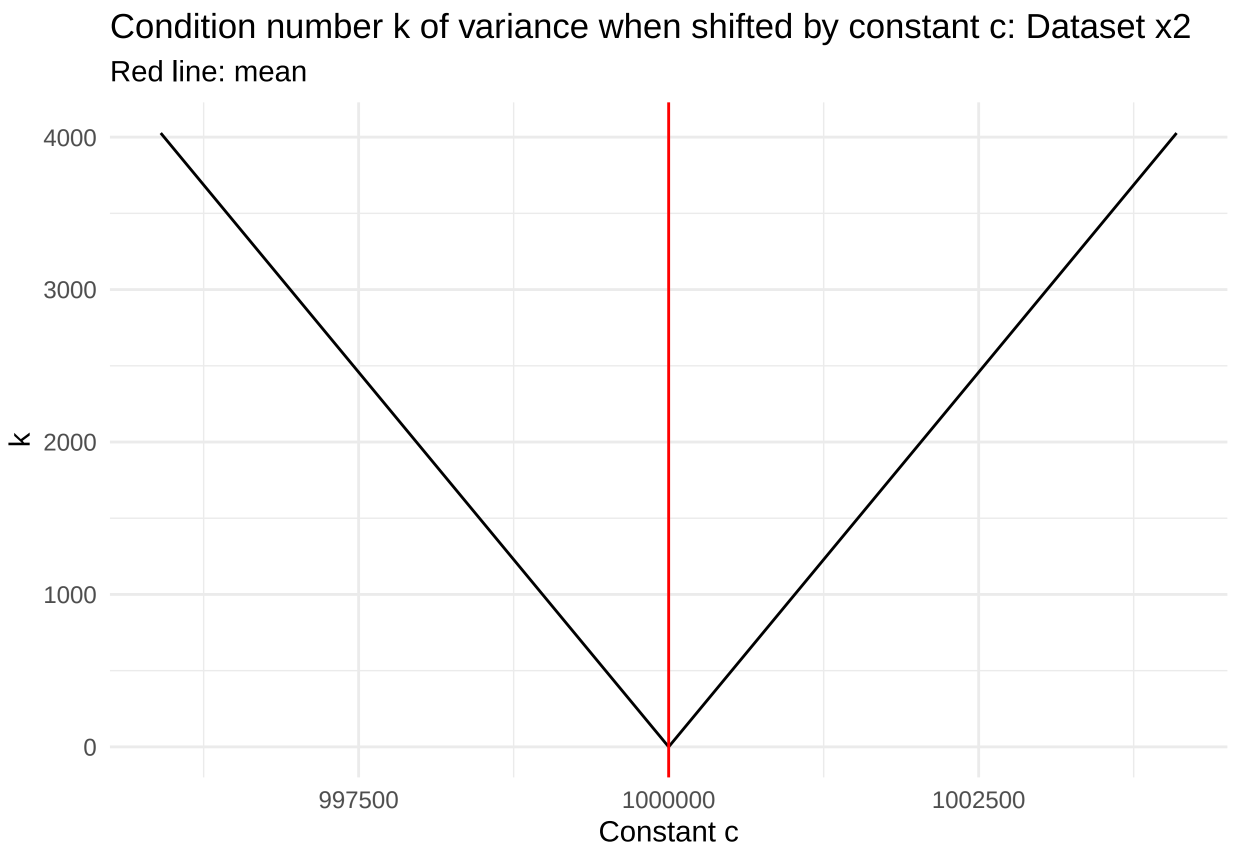
```
show_shifted_k(x1, ": Dataset x1")
```



```
con_num_variance_shift(x2, 1000000)
```

```
## [1] 1.004071
```

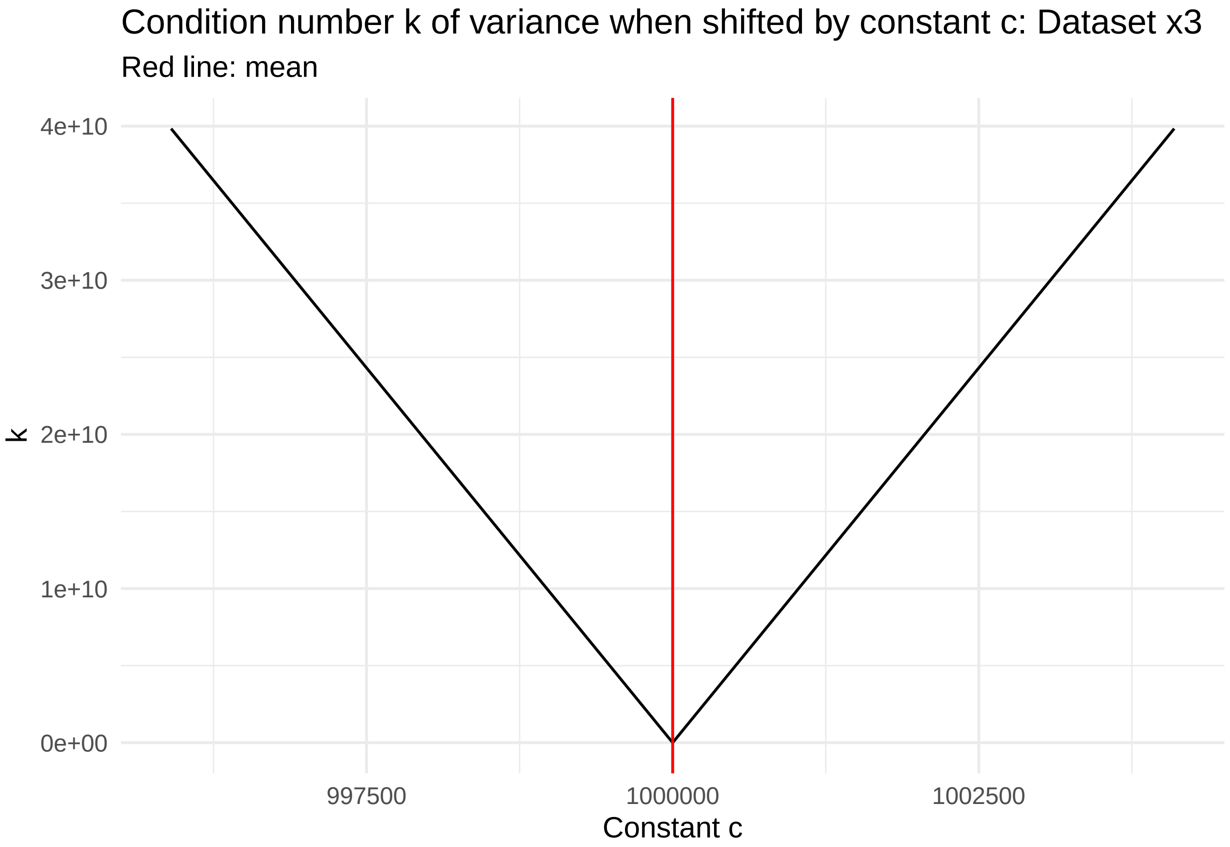
```
show_shifted_k(x2, ": Dataset x2")
```



```
con_num_variance_shift(x3, 1e6)
```

```
## [1] 1.000023
```

```
show_shifted_k(x3, ": Dataset x3")
```



I created a funciton that computes the mean for a sample. It then generates 48 constants on an exponential scale around the mean, which will illustrate the influence of the shifting constants on the condition number. I apply the condition number formula and plot the results on a line chart for each sample. The red lines mark the samples' means.

When plotting the condition number for the shifted algorithm for different constant values of c, it can be observed that the condition number is lowest when c is equal to the sample mean.

When comparing the plots on samples 2 and 3, you may notice that while their condition numbers may both be converging to 1 at around their mean, the condition number of sample 3 rises much faster the larger the difference between the mean and the shifting constant c becomes.